AD-A215 183

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER / 2. GOVT ACCESSION NO	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) **Ada Compiler Validation Summary Report:** Elxsi, Elxsi VADS, Version 5.6, Elxsi 6400, under ENIX, 4.3BSD 12.1 (Host & Target), 890712W1.10115	5. TYPE OF REPORT & PERIOD COVERED 12 July 1989 to 12 July 1990 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Wright-Patterson AFB Dayton, OH, USA	8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION AND ADDRESS Wright-Patterson AFB Dayton, OH, USA	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Ada Joint Program Office United States Department of Defense Washington, DC 20301-3081	12. REPORT DATE 13. NUMBER OF PAGES
14. MONITORING AGENCY NAME & ADDRESS (If different from Controlling Office) Wright-Patterson AFB Dayton, OH, USA	15. SECURITY CLASS (of this report) UNCLASSIFIED 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

UNCLASSIFIED

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Ada Programming language, Ada Compiler Validation Summary Report, Ada Compiler Validation Capability, ACVC, Validation Testing, Ada Validation Office, AVO, Ada Validation Facility, AVF, ANSI/MIL-STD-1815A, Ada Joint Program Office, AJPO

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

Elxsi, Elxsi VADS, Version 5.6, Elxsi 6400, Wright-Patterson AFB, Elxsi 6400 under ENIX, 4.3BSD 12.1(Host & Target), ACVC 1.10.

DD FORM 1473 EDITION OF 1 NOV 65 IS OBSOLETE
1 JAN 73 S/N 0102-LF-014-6601

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

89 11 30 060

AVF Control Number: AVF-VSR-307.0789
89-03-17-ELX

Ada COMPILER
VALIDATION SUMMARY REPORT:
Certificate Number: 890712W1.10115
Elxsi
Elxsi VADS, Version 5.6
Elxsi 6400
under ENIX, 4.3BSD 12.1

Completion of On-Site Testing:
12 July 1989

Prepared By:
Ada Validation Facility
ASD/SCEL
Wright-Patterson AFB OH 45433-6503

Prepared For:
Ada Joint Program Office
United States Department of Defense
Washington DC 20301-3081

Ada Compiler Validation Summary Report:

Compiler Name: Elxsi VADS, Version 5.6

Certificate Number: 890712W1.10115

Host: Elxsi 6400 under
ENIX, 4.3BSD 12.1

Target: Elxsi 6400 under
ENIX, 4.3BSD 12.1

Testing Completed 12 July 1989 Using ACVC 1.10

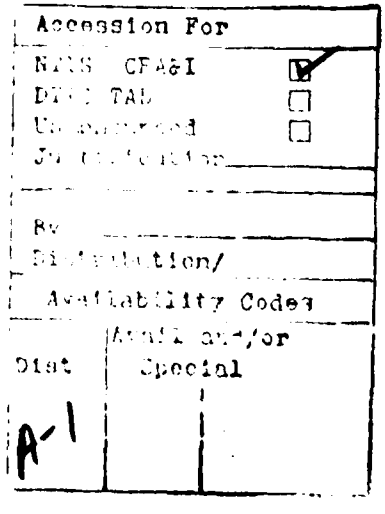
This report has been reviewed and is approved.

Ada Validation Facility
Steve P. Wilson
Technical Director
ASD/SCEL
Wright-Patterson AFB OH 45433-6503

Ada Validation Organization
Dr. John F. Kramer
Institute for Defense Analyses
Alexandria VA 22311

Ada Joint Program Office
Dr. John Solomond
Director
Department of Defense
Washington DC 20301

TABLE OF CONTENTS



CHAPTER 1

INTRODUCTION

This Validation Summary Report (VSR) describes the extent to which a specific Ada compiler conforms to the Ada Standard, ANSI/MIL-STD-1815A. This report explains all technical terms used within it and thoroughly reports the results of testing this compiler using the Ada Compiler Validation Capability (ACVC). An Ada compiler must be implemented according to the Ada Standard, and any implementation-dependent features must conform to the requirements of the Ada Standard. The Ada Standard must be implemented in its entirety, and nothing can be implemented that is not in the Standard.

Even though all validated Ada compilers conform to the Ada Standard, it must be understood that some differences do exist between implementations. The Ada Standard permits some implementation dependencies--for example, the maximum length of identifiers or the maximum values of integer types. Other differences between compilers result from the characteristics of particular operating systems, hardware, or implementation strategies. All the dependencies observed during the process of testing this compiler are given in this report.

The information in this report is derived from the test results produced during validation testing. The validation process includes submitting a suite of standardized tests, the ACVC, as inputs to an Ada compiler and evaluating the results. The purpose of validating is to ensure conformity of the compiler to the Ada Standard by testing that the compiler properly implements legal language constructs and that it identifies and rejects illegal language constructs. The testing also identifies behavior that is implementation-dependent but is permitted by the Ada Standard. Six classes of tests are used. These tests are designed to perform checks at compile time, at link time, and during execution.

INTRODUCTION

1.1 PURPOSE OF THIS VALIDATION SUMMARY REPORT

This VSR documents the results of the validation testing performed on an Ada compiler. Testing was carried out for the following purposes:

- To attempt to identify any language constructs supported by the compiler that do not conform to the Ada Standard
- To attempt to identify any language constructs not supported by the compiler but required by the Ada Standard
- To determine that the implementation-dependent behavior is allowed by the Ada Standard

Testing of this compiler was conducted by SofTech, Inc. under the direction of the AVF according to procedures established by the Ada Joint Program Office and administered by the Ada Validation Organization (AVO). On-site testing was completed 12 July 1989 at San Jose CA.

1.2 USE OF THIS VALIDATION SUMMARY REPORT

Consistent with the national laws of the originating country, the AVO may make full and free public disclosure of this report. In the United States, this is provided in accordance with the "Freedom of Information Act" (5 U.S.C.#552). The results of this validation apply only to the computers, operating systems, and compiler versions identified in this report.

The organizations represented on the signature page of this report do not represent or warrant that all statements set forth in this report are accurate and complete, or that the subject compiler has no nonconformities to the Ada Standard other than those presented. Copies of this report are available to the public from:

Ada Information Clearinghouse
Ada Joint Program Office
OUSDRE
The Pentagon, Rm 3D-139 (Fern Street)
Washington DC 20301-3081

or from:

Ada Validation Facility
ASD/SCEL
Wright-Patterson AFB OH 45433-6503

Questions regarding this report or the validation test results should be directed to the AVF listed above or to:

Ada Validation Organization
Institute for Defense Analyses
1801 North Beauregard Street
Alexandria VA 22311

1.3 REFERENCES

1. Reference Manual for the Ada Programming Language, ANSI/MIL-STD-1815A, February 1983 and ISO 8652-1987.

2. Ada Compiler Validation Procedures and Guidelines, Ada Joint Program Office, 1 January 1987.

3. Ada Compiler Validation Capability Implementers' Guide, SofTech, Inc., December 1986.

4. Ada Compiler Validation Capability User's Guide, December 1986.

1.4 DEFINITION OF TERMS

ACVC	The Ada Compiler Validation Capability. The set of Ada programs that tests the conformity of an Ada compiler to the Ada programming language.
Ada Commentary	An Ada Commentary contains all information relevant to the point addressed by a comment on the Ada Standard. These comments are given a unique identification number having the form AI-ddddd.
Ada Standard	ANSI/MIL-STD-1815A, February 1983 and ISO 8652-1987.
Applicant	The agency requesting validation.
AVF	The Ada Validation Facility. The AVF is responsible for conducting compiler validations according to procedures contained in the Ada Compiler Validation Procedures and Guidelines.
AVO	The Ada Validation Organization. The AVO has oversight authority over all AVF practices for the purpose of maintaining a uniform process for validation of Ada compilers. The AVO provides administrative and technical support for Ada validations to ensure consistent practices.
Compiler	A processor for the Ada language. In the context of this report, a compiler is any language processor, including

cross-compilers, translators, and interpreters.

Failed test — An ACVC test for which the compiler generates a result that demonstrates nonconformity to the Ada Standard.

Host — The computer on which the compiler resides.

Inapplicable test — An ACVC test that uses features of the language that a compiler is not required to support or may legitimately support in a way other than the one expected by the test.

Passed test — An ACVC test for which a compiler generates the expected result.

Target — The computer for which a compiler generates code.

Test — A program that checks a compiler's conformity regarding a particular feature or a combination of features to the Ada Standard. In the context of this report, the term is used to designate a single test, which may comprise one or more files.

Withdrawn test — An ACVC test found to be incorrect and not used to check conformity to the Ada Standard. A test may be incorrect because it has an invalid test objective, fails to meet its test objective, or contains illegal or erroneous use of the language.

1.5 ACVC TEST CLASSES

Conformity to the Ada Standard is measured using the ACVC. The ACVC contains both legal and illegal Ada programs structured into six test classes: A, B, C, D, E, and L. The first letter of a test name identifies the class to which it belongs. Class A, C, D, and E tests are executable, and special program units are used to report their results during execution. Class B tests are expected to produce compilation errors. Class L tests are expected to produce compilation or link errors because of the way in which a program library is used at link time.

Class A tests ensure the successful compilation of legal Ada programs with certain language constructs which cannot be verified at compile time. There are no explicit program components in a Class A test to check semantics. For example, a Class A test checks that reserved words of another language (other than those already reserved in the Ada language) are not treated as reserved words by an Ada compiler. A Class A test is passed if no errors are detected at compile time and the program executes to produce a PASSED message.

Class B tests check that a compiler detects illegal language usage. Class B tests are not executable. Each test in this class is compiled and the resulting compilation listing is examined to verify that every syntax or semantic error in the test is detected. A Class B test is passed if every

illegal construct that it contains is detected by the compiler.

Class C tests check the run time system to ensure that legal Ada programs can be correctly compiled and executed. Each Class C test is self-checking and produces a PASSED, FAILED, or NOT APPLICABLE message indicating the result when it is executed.

Class D tests check the compilation and execution capacities of a compiler. Since there are no capacity requirements placed on a compiler by the Ada Standard for some parameters--for example, the number of identifiers permitted in a compilation or the number of units in a library--a compiler may refuse to compile a Class D test and still be a conforming compiler. Therefore, if a Class D test fails to compile because the capacity of the compiler is exceeded, the test is classified as inapplicable. If a Class D test compiles successfully, it is self-checking and produces a PASSED or FAILED message during execution.

Class E tests are expected to execute successfully and check implementation-dependent options and resolutions of ambiguities in the Ada Standard. Each Class E test is self-checking and produces a NOT APPLICABLE, PASSED, or FAILED message when it is compiled and executed. However, the Ada Standard permits an implementation to reject programs containing some features addressed by Class E tests during compilation. Therefore, a Class E test is passed by a compiler if it is compiled successfully and executes to produce a PASSED message, or if it is rejected by the compiler for an allowable reason.

Class L tests check that incomplete or illegal Ada programs involving multiple, separately compiled units are detected and not allowed to execute. Class L tests are compiled separately and execution is attempted. A Class L test passes if it is rejected at link time--that is, an attempt to execute the main program must generate an error message before any declarations in the main program or any units referenced by the main program are elaborated. In some cases, an implementation may legitimately detect errors during compilation of the test.

Two library units, the package REPORT and the procedure CHECK_FILE, support the self-checking features of the executable tests. The package REPORT provides the mechanism by which executable tests report PASSED, FAILED, or NOT APPLICABLE results. It also provides a set of identity functions used to defeat some compiler optimizations allowed by the Ada Standard that would circumvent a test objective. The procedure CHECK_FILE is used to check the contents of text files written by some of the Class C tests for chapter 14 of the Ada Standard. The operation of REPORT and CHECK_FILE is checked by a set of executable tests. These tests produce messages that are examined to verify that the units are operating correctly. If these units are not operating correctly, then the validation is not attempted.

The text of each test in the ACVC follows conventions that are intended to ensure that the tests are reasonably portable without modification. For example, the tests make use of only the basic set of 55 characters, contain lines with a maximum length of 72 characters, use small numeric values, and place features that may not be supported by all implementations in separate

tests. However, some tests contain values that require the test to be customized according to implementation-specific values--for example, an illegal file name. A list of the values used for this validation is provided in Appendix C.

A compiler must correctly process each of the tests in the suite and demonstrate conformity to the Ada Standard by either meeting the pass criteria given for the test or by showing that the test is inapplicable to the implementation. The applicability of a test to an implementation is considered each time the implementation is validated. A test that is inapplicable for one validation is not necessarily inapplicable for a subsequent validation. Any test that was determined to contain an illegal language construct or an erroneous language construct is withdrawn from the ACVC and, therefore, is not used in testing a compiler. The tests withdrawn at the time of this validation are given in Appendix D.

CHAPTER 2

CONFIGURATION INFORMATION

2.1 CONFIGURATION TESTED

The candidate compilation system for this validation was tested under the following configuration:

Compiler: Elxsi VADS, Version 5.6

ACVC Version: 1.10

Certificate Number: 890712W1.10115

Host Computer:

Machine:	Elxsi 6400
Operating System:	ENIX, 4.3BSD 12.1
Memory Size:	64 Megabytes

Target Computer:

Machine:	Elxsi 6400
Operating System:	ENIX, 4.3BSD 12.1
Memory Size:	64 Megabytes

2.2 IMPLEMENTATION CHARACTERISTICS

One of the purposes of validating compilers is to determine the behavior of a compiler in those areas of the Ada Standard that permit implementations to differ. Class D and E tests specifically check for such implementation differences. However, tests in other classes also characterize an implementation. The tests demonstrate the following characteristics:

a. Capacities.

(1) The compiler correctly processes a compilation containing 723 variables in the same declarative part. (See test D29002K.)

(2) The compiler correctly processes tests containing loop statements nested to 65 levels. (See tests D55A03A..H (8 tests).)

(3) The compiler correctly processes tests containing block statements nested to 65 levels. (See test D56001B.)

(4) The compiler correctly processes tests containing recursive procedures separately compiled as subunits nested to 17 levels. (See tests D64005E..G (3 tests).)

b. Predefined types.

(1) This implementation supports the additional predefined type LONG_FLOAT in package STANDARD. (See tests B86001T..Z (7 tests).)

c. Expression evaluation.

The order in which expressions are evaluated and the time at which constraints are checked are not defined by the language. While the ACVC tests do not specifically attempt to determine the order of evaluation of expressions, test results indicate the following:

(1) None of the default initialization expressions for record components are evaluated before any value is checked for membership in a component's subtype. (See test C32117A.)

(2) Assignments for subtypes are performed with the same precision as the base type. (See test C35712B.)

(3) This implementation uses no extra bits for extra precision and uses all extra bits for extra range. (See test C35903A.)

(4) Sometimes NUMERIC_ERROR is raised when an integer literal operand in a comparison or membership test is outside the range of the base type. (See test C45232A.)

(5) Sometimes NUMERIC_ERROR is raised when a literal operand in a fixed-point comparison or membership test is outside the range of the base type. (See test C45252A.)

(6) Underflow is gradual. (See tests C45524A..Z.)

d. Rounding.

The method by which values are rounded in type conversions is not defined by the language. While the ACVC tests do not specifically attempt to determine the method of rounding, the test results indicate the following:

(1) The method used for rounding to integer is round to even. (See tests C46012A..Z.)

(2) The method used for rounding to longest integer is round to even. (See tests C46012A..Z.)

(3) The method used for rounding to integer in static universal real expressions is round to even. (See test C4A014A.)

e. Array types.

An implementation is allowed to raise NUMERIC_ERROR or CONSTRAINT_ERROR for an array having a 'LENGTH that exceeds STANDARD.INTEGER'LAST and/or SYSTEM.MAX_INT.

For this implementation:

(1) Declaration of an array type or subtype declaration with more than SYSTEM.MAX_INT components raises no exception. (See test C36003A.)

(2) NUMERIC_ERROR is raised when a null array type with INTEGER'LAST + 2 components is declared. (See test C36202A.)

(3) NUMERIC_ERROR is raised when a null array type with SYSTEM.MAX_INT + 2 components is declared. (See test C36202B.)

(4) A packed BOOLEAN array having a 'LENGTH exceeding INTEGER'LAST raises NUMERIC_ERROR when the array type is declared. (See test C52103X.)

(5) A packed two-dimensional BOOLEAN array with more than INTEGER'LAST components raises NUMERIC_ERROR when the array type is declared. (See test C52104Y.)

(6) A null array with one dimension of length greater than INTEGER'LAST may raise NUMERIC_ERROR or CONSTRAINT_ERROR either when declared or assigned. Alternatively, an implementation may accept the declaration. However, lengths must match in array slice assignments. This implementation raises NUMERIC_ERROR when the array type is declared. (See test E52103Y.)

(7) In assigning one-dimensional array types, the expression is evaluated in its entirety before CONSTRAINT_ERROR is raised when checking whether the expression's subtype is compatible with the target's subtype. (See test C52013A.)

(8) In assigning two-dimensional array types, the expression is not evaluated in its entirety before CONSTRAINT_ERROR is raised when checking whether the expression's subtype is compatible with the target's subtype. (See test C52013A.)

f. Discriminated types.

(1) In assigning record types with discriminants, the expression is evaluated in its entirety before CONSTRAINT_ERROR is raised when checking whether the expression's subtype is compatible with the target's subtype. (See test C52013A.)

g. Aggregates.

(1) In the evaluation of a multi-dimensional aggregate, all choices are evaluated before checking against the index type. (See tests C43207A and C43207B.)

(2) In the evaluation of an aggregate containing subaggregates, all choices are evaluated before being checked for identical bounds. (See test E43212B.)

(3) CONSTRAINT_ERROR is raised after all choices are evaluated when a bound in a non-null range of a non-null aggregate does not belong to an index subtype. (See test E43211B.)

h. Pragmas.

(1) The pragma INLINE is supported for functions and procedures. (See tests LA3004A..B, EA3004C..D, and CA3004E..F.)

i. Generics

(1) Generic specifications and bodies can be compiled in separate compilations. (See tests CA1012A, CA2009C, CA2009F, BC3204C, and BC3205D.)

(2) Generic unit bodies and their subunits can be compiled in separate compilations. (See test CA3011A.)

j. Input and output

(1) The package SEQUENTIAL_IO can be instantiated with unconstrained array types and record types with discriminants without defaults. (See tests AE2101C, EE2201D, and EE2201E.)

(2) The package DIRECT_IO can be instantiated with unconstrained array types and record types with discriminants without defaults. (See tests AE2101H, EE2401D, and EE2401G.)

(3) Modes IN_FILE and OUT_FILE are supported for SEQUENTIAL_IO. (See tests CE2102D..E, CE2102N, and CE2102P.)

(4) Modes IN_FILE, OUT_FILE, and INOUT_FILE are supported for DIRECT_IO. (See tests CE2102F, CE2102I..J, CE2102R, CE2102T, and CE2102V.)

(5) Modes IN_FILE and OUT_FILE are supported for text files. (See tests CE3102E and CE3102I..K.)

(6) RESET and DELETE operations are supported for SEQUENTIAL_IO. (See tests CE2102G and CE2102X.)

(7) RESET and DELETE operations are supported for DIRECT_IO. (See tests CE2102K and CE2102Y.)

(8) RESET and DELETE operations are supported for text files. (See tests CE3102F..G, CE3104C, CE3110A, and CE3114A.)

(9) Overwriting to a sequential file truncates to the last element written. (See test CE2208B.)

(10) Temporary sequential files are given names and deleted when closed. (See test CE2108A.)

(11) Temporary direct files are given names and deleted when closed. (See test CE2108C.)

(12) Temporary text files are given names and deleted when closed. (See test CE3112A.)

(13) More than one internal file can be associated with each external file for sequential files when writing or reading. (See tests CE2107A..E, CE2102L, CE2110B, and CE2111D.)

(14) More than one internal file can be associated with each external file for direct files when writing or reading. (See tests CE2107F..H (3 tests), CE2110D, and CE2111H.)

(15) More than one internal file can be associated with each external file for text files when reading or writing. (See tests CE3111A..E, CE3114B, and CE3115A.)

CHAPTER 3

TEST INFORMATION

3.1 TEST RESULTS

Version 1.10 of the ACVC comprises 3717 tests. When this compiler was tested, 44 tests had been withdrawn because of test errors. The AVF determined that 348 tests were inapplicable to this implementation. All inapplicable tests were processed during validation testing except for 187 executable tests that use floating-point precision exceeding that supported by the implementation. Modifications to the code, processing, or grading for 11 tests were required to successfully demonstrate the test objective. (See section 3.6.)

The AVF concludes that the testing results demonstrate acceptable conformity to the Ada Standard.

3.2 SUMMARY OF TEST RESULTS BY CLASS

RESULT	TEST CLASS						TOTAL
	A	B	C	D	E	L	
Passed	129	1128	1977	17	28	46	3325
Inapplicable	0	10	338	0	0	0	348
Withdrawn	1	2	35	0	6	0	44
TOTAL	130	1140	2350	17	34	46	3717

3.3 SUMMARY OF TEST RESULTS BY CHAPTER

RESULT	CHAPTER													TOTAL
	2	3	4	5	6	7	8	9	10	11	12	13	14	
Passed	198	577	533	242	172	99	159	331	137	36	252	290	299	3325
Inappl	14	72	147	6	0	0	7	1	0	0	0	79	22	348
Wdrn	1	1	0	0	0	0	0	2	0	0	1	35	4	44
TOTAL	213	650	680	248	172	99	166	334	137	36	253	404	325	3717

3.4 WITHDRAWN TESTS

The following 44 tests were withdrawn from ACVC Version 1.10 at the time of this validation:

E28005C	A39005G	B97102E	C97116A	BC3009B	CD2A62D
CD2A63A	CD2A63B	CD2A63C	CD2A63D	CD2A66A	CD2A66B
CD2A66C	CD2A66D	CD2A73A	CD2A73B	CD2A73C	CD2A73D
CD2A76A	CD2A76B	CD2A76C	CD2A76D	CD2A81G	CD2A83G
CD2A84M	CD2A84N	CD2B15C	CD2D11B	CD5007B	CD50110
ED7004B	ED7005C	ED7005D	ED7006C	ED7006D	CD7105A
CD7203B	CD7204B	CD7205C	CD7205D	CE2107I	CE3111C
CE3301A	CE3411B				

See Appendix D for the reason that each of these tests was withdrawn.

3.5 INAPPLICABLE TESTS

Some tests do not apply to all compilers because they make use of features that a compiler is not required by the Ada Standard to support. Others may depend on the result of another test that is either inapplicable or withdrawn. The applicability of a test to an implementation is considered each time a validation is attempted. A test that is inapplicable for one validation attempt is not necessarily inapplicable for a subsequent attempt. For this validation attempt, 348 tests were inapplicable for the reasons indicated:

a. C24113L..Y (14 tests) are not applicable because they have lines which exceed the maximum line length of this implementation.

b. The following 187 tests are not applicable because they have floating-point type declarations requiring more digits than SYSTEM.MAX_DIGITS:

C35705L..Y	C35706L..Y	C35707L..Y	C35708L..Y
C35802L..Z	C45241L..Y	C45321L..Y	C45421L..Y
C45521L..Z	C45524L..Z	C45621L..Z	C45641L..Y
C46012L..Z			

c. C35702A and B86001T are not applicable because this implementation supports no predefined type SHORT_FLOAT.

d. The following 16 tests are not applicable because this implementation does not support a predefined type SHORT_INTEGER:

C45231B	C45304B	C45502B	C45503B	C45504B
C45504E	C45611B	C45613B	C45614B	C45631B
C45632B	B52004E	C55B07B	B55B09D	B86001V
CD7101E				

e. The following 16 tests are not applicable because this implementation does not support a predefined type LONG_INTEGER:

C45231C	C45304C	C45502C	C45503C	C45504C
C45504F	C45611C	C45613C	C45614C	C45631C
C45632C	B52004D	C55B07A	B55B09C	B86001W
CD7101F				

f. C45231D, B86001X, and CD7101G are not applicable because this implementation does not support any predefined integer type with a name other than INTEGER, LONG_INTEGER, or SHORT_INTEGER.

g. C45531M..P (4 tests) and C45532M..P (4 tests) are not applicable because the value of SYSTEM.MAX_MANTISSA is less than 47.

h. C86001F is not applicable because, for this implementation, the package TEXT_IO is dependent upon package SYSTEM. These tests recompile package SYSTEM, making package TEXT_IO, and hence package REPORT, obsolete.

i. B86001Y is not applicable because this implementation supports no predefined fixed-point type other than DURATION.

j. B86001Z is not applicable because this implementation does not support any predefined float type with a name other than FLOAT, SHORT_FLOAT, or LONG_FLOAT.

k. C96005B is not applicable because there are no values of type DURATION'BASE that are outside the range of DURATION.

l. CD1009C, CD2A41A..B (2 tests), CD2A41E, and CD2A42A..J (10 tests) are not applicable because this implementation does not support size clauses for floating point types which use less than 32 bits.

m. CD2A61I and CD2A61J are not applicable because this implementation does not support size clauses for array types, which imply compression, with component types of composite or floating point

types. This implementation requires an explicit size clause on the component type.

n. CD2A84B..I (8 tests) and CD2A84K..L (2 tests) are not applicable because this implementation does not support size clauses for access types which use less than 32 bits.

o. CD2A91A..E (5 tests), CD5012J, CD5013S, and CD5014S are not applicable because this implementation does not support size clauses for tasks or task types.

p. The following 42 tests are not applicable because this implementation does not support an address clause when a dynamic address is applied to a variable requiring initialization:

CD5003B..H	CD5011A..H	CD5011L..N	CD5011Q
CD5011R	CD5012A..I	CD5012L	CD5013B
CD5013D	CD5013F	CD5013H	CD5013L
CD5013N	CD5013R	CD5014T..X	

q. CE2102D is inapplicable because this implementation supports CREATE with IN_FILE mode for SEQUENTIAL_IO.

r. CE2102E is inapplicable because this implementation supports CREATE with OUT_FILE mode for SEQUENTIAL_IO.

s. CE2102F is inapplicable because this implementation supports CREATE with INOUT_FILE mode for DIRECT_IO.

t. CE2102I is inapplicable because this implementation supports CREATE with IN_FILE mode for DIRECT_IO.

u. CE2102J is inapplicable because this implementation supports CREATE with OUT_FILE mode for DIRECT_IO.

v. CE2102N is inapplicable because this implementation supports OPEN with IN_FILE mode for SEQUENTIAL_IO.

w. CE2102O is inapplicable because this implementation supports RESET with IN_FILE mode for SEQUENTIAL_IO.

x. CE2102P is inapplicable because this implementation supports OPEN with OUT_FILE mode for SEQUENTIAL_IO.

y. CE2102Q is inapplicable because this implementation supports RESET with OUT_FILE mode for SEQUENTIAL_IO.

z. CE2102R is inapplicable because this implementation supports OPEN with INOUT_FILE mode for DIRECT_IO.

aa. CE2102S is inapplicable because this implementation supports RESET with INOUT_FILE mode for DIRECT_IO.

ab. CE2102T is inapplicable because this implementation supports OPEN with IN_FILE mode for DIRECT_IO.

ac. CE2102U is inapplicable because this implementation supports RESET with IN_FILE mode for DIRECT_IO.

ad. CE2102V is inapplicable because this implementation supports OPEN with OUT_FILE mode for DIRECT_IO.

ae. CE2102W is inapplicable because this implementation supports RESET with OUT_FILE mode for DIRECT_IO.

af. CE3102E is inapplicable because this implementation supports CREATE with IN_FILE mode for text files.

ag. CE3102F is inapplicable because this implementation supports RESET for text files.

ah. CE3102G is inapplicable because this implementation supports deletion of an external file for text files.

ai. CE3102I is inapplicable because this implementation supports CREATE with OUT_FILE mode for text files.

aj. CE3102J is inapplicable because this implementation supports OPEN with IN_FILE mode for text files.

ak. CE3102K is inapplicable because this implementation supports OPEN with OUT_FILE mode for text files.

al. CE3115A is not applicable because resetting of an external file with OUT_FILE mode is not supported with multiple internal files associated with the same external file when they have different modes.

3.6 TEST, PROCESSING, AND EVALUATION MODIFICATIONS

It is expected that some tests will require modifications of code, processing, or evaluation in order to compensate for legitimate implementation behavior. Modifications are made by the AVF in cases where legitimate implementation behavior prevents the successful completion of an (otherwise) applicable test. Examples of such modifications include: adding a length clause to alter the default size of a collection; splitting a Class B test into subtests so that all errors are detected; and confirming that messages produced by an executable test demonstrate conforming behavior that wasn't anticipated by the test (such as raising one exception instead of another).

Modifications were required for 11 tests.

The following 10 tests were split because syntax errors at one point resulted in the compiler not detecting other errors in the test:

B24009A	B33301B	B38003A	B38003B	B38009A	B38009B
B41202A	B91001H	BC1303F	BC3005B		

CD2A83A required modification to the collection size. The original size was 256 which caused the exception STORAGE_ERROR to be raised by this implementation. The collection size was modified to 296 and the test produced a passed result. The reason for a larger collection size is that this implementation uses 32 bytes (STORAGE_UNITS) for its array descriptors, 8 bytes each for: the low bound; the high bound; the array length; and the component size. There are 8 allocations in this ACVC test, each array is 3 to 5 bytes in size, so the total amount of space needed is roughly 8*(32+5)=296.

3.7 ADDITIONAL TESTING INFORMATION

3.7.1 Prevalidation

Prior to validation, a set of test results for ACVC Version 1.10 produced by the Elxsi VADS was submitted to the AVF by the applicant for review. Analysis of these results demonstrated that the compiler successfully passed all applicable tests, and the compiler exhibited the expected behavior on all inapplicable tests.

3.7.2 Test Method

Testing of the Elxsi VADS using ACVC Version 1.10 was conducted on-site by a validation team from the AVF. The configuration in which the testing was performed is described by the following designations of hardware and software components:

Host computer:	Elxsi 6400
Host operating system:	ENIX, 4.3BSD 12.1
Target computer:	Elxsi 6400
Target operating system:	ENIX, 4.3BSD 12.1
Compiler:	Elxsi VADS, Version 5.6

A magnetic tape containing all tests except for withdrawn tests and tests requiring unsupported floating-point precisions was taken on-site by the validation team for processing. Tests that make use of implementation-specific values were customized before being written to the magnetic tape. Tests requiring modifications during the prevalidation testing were included in their modified form on the magnetic tape.

The contents of the magnetic tape were loaded directly onto the host computer.

After the test files were loaded to disk, the full set of tests was compiled, linked, and all executable tests were run on the Elxsi 6400. Results were printed from the from the host computer.

The compiler was tested using command scripts provided by Elxsi and reviewed by the validation team. The compiler was tested using all default option settings except for the following:

OPTION	EFFECT
-el	Produce error listing if any errors or warnings occur.
-u	Link with same name as file name (for executable tests).

Tests were compiled, linked, and executed (as appropriate) using a single host computer. Test output, compilation listings, and job logs were captured on magnetic tape and archived at the AVF. The listings examined on-site by the validation team were also archived.

3.7.3 Test Site

Testing was conducted at San Jose CA and was completed on 12 July 1989.

APPENDIX A

DECLARATION OF CONFORMANCE

Elxsi has submitted the following Declaration of Conformance concerning the Elxsi VADS.

DECLARATION OF CONFORMANCE

Compiler Implementor: Elxsi
Ada Validation Facility: ASD/SCEL, Wright-Patterson AFB OH 45433-6503
Ada Compiler Validation Capability (ACVC) Version: 1.10

Base Configuration

Base Compiler Name:	Elxsi VADS	Version: 5.6	
Host Architecture ISA:	Elxsi 6400	OS & VER#:	ENIX 4.3BSD 12.1
Target Architecture ISA:	Elxsi 6400	OS & VER#:	ENIX 4.3BSD 12.1

Implementor's Declaration

I, the undersigned, representing Elxsi have implemented no deliberate extensions to the Ada Language Standard ANSI/MIL-STD-1815A in the compiler(s) listed in this declaration. I declare that Elxsi is the owner of record of the Ada language compiler listed above and as such, is responsible for maintaining said compiler in conformance to ANSI/MIL-STD-1815A. All certificates and registrations for Ada language compiler listed in this declaration shall be made in the owner's corporate name.

Date: 7/12/89

Elxsi
Kirk Erickson, Member of Technical Staff

Owner's Declaration

I, the undersigned, representing ELXSI, take full responsibility for implementation and maintenance of the Ada compilers listed above, and agree to the public disclosure of the final Validation Summary Report. I declare that all of the Ada language compilers listed, and their host/target performance are in compliance with the Ada Language Standard ANSI/MIL-STD-1815A.

Date: 7/12/89

Elxsi
Kirk Erickson, Member of Technical Staff

APPENDIX B

APPENDIX F OF THE Ada STANDARD

The only allowed implementation dependencies correspond to implementation-dependent pragmas, to certain machine-dependent conventions as mentioned in chapter 13 of the Ada Standard, and to certain allowed restrictions on representation clauses. The implementation-dependent characteristics of the Elxsi VADS, Version 5.6, as described in this Appendix, are provided by Elxsi. Unless specifically noted otherwise, references in this Appendix are to compiler documentation and not to this report. Implementation-specific portions of the package STANDARD, which are not a part of Appendix F, are:

```
package STANDARD is

   ...

   type INTEGER is range -9223372036854775808..9223372036854775807;

   type FLOAT is digits 6 range -2#1.11111111111111111111111#E127..
                                 2#1.11111111111111111111111#E127;

   type LONG_FLOAT is digits 15 range
     -2#1.11111111111111111111111111111111111111111111111111111111#E1023..
      2#1.11111111111111111111111111111111111111111111111111111111#E1023;

   type DURATION is delta 0.001 range -2147483.648..2147483.647;

   ...

end STANDARD;
```

F.1. Predefined Types

The STANDARD.INTEGER type is 64-bits and ranges from

-9_223_372_036_854_775_808 .. 9_223_372_036_854_775_807

The package SYSTEM also defines the following integer types:

8-bit TINY_INT
16-bit SHORT_INT
32-bit INT

Two floating point types are provided:

FLOAT
which has 32 bits and 6 digits of precision; and

LONG_FLOAT
which has 64 bits and 15 digits of precision.

VADS Ada provides fixed point types mapped to the 64-bit INTEGER type. The type DURATION is defined as

delta 0.001 range -2147483.648 .. 2147483.647;

The image of a character that is not a graphic character is defined to be the corresponding 2 or 3 character identifier from package ASCII of RM Annex C-4.

F.2. Predefined Pragmas

All the predefined pragmas are recognized by the implementation, and behave as described in Appendix B of the RM with the following restrictions.

The pragmas CONTROLLED, MEMORY_SIZE, OPTIMIZE, SHARED, STORAGE_UNIT and SYSTEM_NAME have no effect.

PRAGMA INLINE limits recursive calls that can be expanded with the pragma up to a maximum depth of 8.

PRAGMA INTERFACE supports calls to the language names C, FORTRAN, SYSTEM and UNCHECKED. For C, the types of parameters and the result type for functions must be scalar, access or SYSTEM.ADDRESS. For FORTRAN, all parameters are passed by reference; the parameter types must be SYSTEM.ADDRESS. The result type for a FORTRAN function must be a scalar type. The SYSTEM language name is used to call System Foundation intrinsics; all parameters are passed by value, and parameter/result types must be scalar, access, or SYSTEM.ADDRESS. The UNCHECKED language name may be used to interface to assembler; the compiler will generate the call as if it were to an Ada procedure.

To specify an exact function name as the linker expects it, use PRAGMA INTERFACE_NAME (defined below); otherwise the link-name is the lower-case version of the Ada name.

PRAGMA PACK will pack array components to bit sizes corresponding to powers of 2 (if the field is smaller than STORAGE_UNIT bits). Objects larger than a single STORAGE_UNIT are packed to the nearest STORAGE_UNIT.

PRAGMA PRIORITY - priorities range from 0 to 7, with 7 the most urgent.

PRAGMA SUPPRESS is supported in the single parameter form. The double parameter form of the pragma with a name of an object, type, or subtype is recognized, but has no effect. DIVISION_CHECK cannot be suppressed.

F.3. Implementation-dependent Pragmas

PRAGMA ALIGN — Can be specified at the beginning of a package; it aligns all data in the package so that it resides in its own page(s). There should not be any initialized data in the aligned package.

PRAGMA EXTERNAL_NAME — Allows the user to specify a *link_name* for an Ada variable or subprogram so that the object can be referenced from other languages using the syntax shown below.

pragma EXTERNAL_NAME (*object_or_subprogram_name, "linker_name"*);

Objects must be variables defined in a package specification; subprograms can be either library level or within a package specification. The *link_name* must be constructed as expected by the linker.

PRAGMA IMPLICIT_CODE — Takes one parameter having the value ON or OFF. Specifies that implicit code generated by the compiler (i.e. prologue and epilogue code that moves parameters on the stack) is allowed (ON) or disallowed (OFF) and is used only within the declarative part of a machine code procedure. A warning is issued if OFF is used and any implicit code needs to be generated. Implicit code is always generated by default.

PRAGMA INLINE_ONLY — Is used in the same way as **pragma INLINE**, except that it indicates to the compiler that the subprogram must *always* be inlined. This pragma also suppresses the generation of a callable version of the routine which saves code space.

PRAGMA INTERFACE_NAME — Allows names defined in another language to be referenced directly in Ada, replacing all occurrences of *ada_name* with an external reference to *link_name* in the object file, using the format shown below. pragma INTERFACE_NAME (*ada_name, "link_name"*);

This pragma is allowed at the place of a declarative item in a package specification and must apply to a name declared earlier in the same package specification.

PRAGMA NO_IMAGE — Suppresses the generation of the image array used for the IMAGE attribute of enumeration types. This eliminates the overhead required to store the array in the executable image. Takes the name of an enumeration type as a parameter, and must be in the same declarative part as the type.

PRAGMA SHARE_CODE — Provides for the sharing of object code between multiple instantiations of the same generic procedure or package body. A 'parent' instantiation is created and subsequent instantiations of the same types can share the parent's object code, reducing program size and compilation times. This pragma is only allowed in the following places: immediately within a declarative part, immediately within a package specification, or after a library unit in a compilation, but before any subsequent compilation unit. The form of this pragma is

pragma SHARE_CODE (*generic_name, boolean_literal*);

The pragma can reference either the generic unit or the instantiated unit. When it references a generic unit, it sets sharing on or off for all instantiations of that generic unless overridden by specific SHARE_CODE pragmas for individual instantiations. When it references an instantiated unit, sharing is on or off only for that unit. Generics are shared by default, except in the following cases:

1) when generic formal types other than integer, enumeration, SYSTEM.ADDRESS or floating point are used;
2) when the generic unit or its actuals use pragma INLINE;
3) when the representations of the actual type parameters are not the same for each of the instantiations;
4) when the generic has a formal *in out* parameter and the subtype of the corresponding actual is not the same as the subtype of the formal parameter.

Sharing generics causes a slight execution time penalty because all type attributes must be indirectly referenced (as if an extra calling argument were added). However, it substantially reduces compilation time in most circumstances and reduces program size.

We have compiled a unit, SHARED_IO, in the **standard** library that instantiates all Ada generic I/O packages for the most commonly used base types. Thus, any instantiation of an Ada I/O generic package will share one of the parent instantiation generic bodies.

F.4. Implementation-dependent Attributes

VADS provides one implementation-dependent attribute, 'REF. There are two forms of use for this attribute, X'REF and SYSTEM.ADDRESS'REF(N).

X'REF is used only in machine_code procedures to generate a reference to the entity to which it is applied. In X'REF, X must be either a constant, variable, procedure, function or label. The attribute returns a value of the type MACHINE_CODE.OPERAND and may only be used to designate an operand within a code-statement.

SYSTEM.ADDRESS'REF(N) can be used anywhere to convert a static universal_integer expression N to an address. Its effect is similar to the effect of an unchecked conversion from integer to address. In particular, it should be used to place an object at an address, using the form

for *object* use at SYSTEM.ADDRESS'REF (*integer_value*);

F.5. Specification of package SYSTEM

```
package SYSTEM
is
      type NAME is ( elxsi_bsd, elxsi_att );
      SYSTEM_NAME       : constant NAME := elxsi_bsd;

      STORAGE_UNIT            : constant := 8;
      MEMORY_SIZE       : constant := 33_554_432; -- our minimum config.

      -- System-Dependent Named Numbers

      MIN_INT                 : constant := -9_223_372_036_854_775_808;
      MAX_INT                 : constant :=  9_223_372_036_854_775_807;
      MAX_DIGITS        : constant := 15;
      MAX_MANTISSA            : constant := 31;
      FINE_DELTA        : constant := 2.0**(-MAX_MANTISSA);
      TICK              : constant := 0.01;

      -- Other System-dependent Declarations

      subtype PRIORITY is INTEGER range 0 .. 7;

      MAX_REC_SIZE : INTEGER := 64*1024;

      type ADDRESS is private;

      NO_ADDR : constant ADDRESS;

      type  TINY_INT is             range -128..127;
      subtype     NATURAL_TINY_INT is    TINY_INT range 0..TINY_INT'last;
      type  UNSIGNED_TINY_INT is    range 0..255;

      type  SHORT_INT is            range -32768..32767;
      subtype     NATURAL_SHORT_INT is   SHORT_INT range 0..SHORT_INT'last;
      type  UNSIGNED_SHORT_INT is   range 0..65535;

      type  INT is                  range -2147483648..2147483647;
      subtype     NATURAL_INT is          INT range 0..INT'last;
      type  UNSIGNED_INT is         range 0..4294967295;

      subtype     LONG_INT is             INTEGER;

      for   TINY_INT'size use             STORAGE_UNIT;
      for   UNSIGNED_TINY_INT'size use    STORAGE_UNIT;
      for   SHORT_INT'size use            2 * STORAGE_UNIT;
      for   UNSIGNED_SHORT_INT'size use   2 * STORAGE_UNIT;
      for   INT'size use                  4 * STORAGE_UNIT;
      for   UNSIGNED_INT'size use         4 * STORAGE_UNIT;

      function PHYSICAL_ADDRESS(I: INTEGER) return ADDRESS;
      function ADDR_GT(A, B: ADDRESS) return BOOLEAN;
      function ADDR_LT(A, B: ADDRESS) return BOOLEAN;
      function ADDR_GE(A, B: ADDRESS) return BOOLEAN;
```

```
        function ADDR_LE(A, B: ADDRESS) return BOOLEAN;
        function ADDR_DIFF(A, B: ADDRESS) return INT;
        function INCR_ADDR(A: ADDRESS; INCR: INT) return ADDRESS;
        function DECR_ADDR(A: ADDRESS; DECR: INT) return ADDRESS;

        function ">"(A, B: ADDRESS) return BOOLEAN renames ADDR_GT;
        function "<"(A, B: ADDRESS) return BOOLEAN renames ADDR_LT;
        function ">="(A, B: ADDRESS) return BOOLEAN renames ADDR_GE;
        function "<="(A, B: ADDRESS) return BOOLEAN renames ADDR_LE;
        function "-"(A, B: ADDRESS) return INT renames ADDR_DIFF;
        function "+"(A: ADDRESS; DECR: INT) return ADDRESS renames INCR_ADDR;
        function "-"(A: ADDRESS; DECR: INT) return ADDRESS renames DECR_ADDR;

        pragma inline(ADDR_GT);
        pragma inline(ADDR_LT);
        pragma inline(ADDR_GE);
        pragma inline(ADDR_LE);
        pragma inline(ADDR_DIFF);
        pragma inline(INCR_ADDR);
        pragma inline(DECR_ADDR);
        pragma inline(PHYSICAL_ADDRESS);

    private

        type ADDRESS is range -(2**31)..(2**31-1);
        for ADDRESS'size use 4 * STORAGE_UNIT;

        NO_ADDR : constant ADDRESS := ADDRESS(0);

    end SYSTEM;
```

F.6. Restrictions on Representation Clauses

Bit-level representation clauses are supported; length clauses are supported; enumeration representation clauses are supported; change of representation is supported. The restrictions on representation clauses follow.

F.6.1. Length Clauses

Length clauses specifying 'Size for floating point types are not accepted (unless the specified value is either 32 or 64). Length clauses specifying 'Size for access types are not accepted (unless the specified value is 32). Length clauses specifying 'Size for array types with composite components are not accepted if compression of the components is required. Length clauses specifying 'Size for task types are not accepted. Length clauses for initialized variables are not permitted. Length clauses specifying 'SMALL must be a power of 2.0.

F.6.2. Record Representation Clauses

Component clauses that span storage units must be aligned on STORAGE_UNIT boundaries. A component that is itself a record must occupy a power of 2 bits.

F.6.3. Address Clauses

Address clauses are only supported for objects.

F.6.4 Representation Attributes

The ADDRESS attribute is only supported for variables, constants, procedures, and functions.

F.7. Conventions for Implementation-generated Names

There are no implementation generated names.

F.8. Interpretation of Expressions in Address Clauses

The implementation-dependent attribute SYSTEM.ADDRESS'REF(n) should be used to create the address value in an address clause. Alternatively, the system.physical_address() routine can be used to create an address value.

F.9. Restrictions on Unchecked Conversions

The predefined generic function UNCHECKED_CONVERSION cannot be instantiated with a target type that is an unconstrained array type or an unconstrained record type with discriminants. A warning will be issued if you try to convert between objects of different sizes.

F.10. Implementation Characteristics of I/O Packages

The only restriction on the types with which DIRECT_IO and SEQUENTIAL_IO can be instantiated is that the element size must be less than a maximum given by the variable SYSTEM.MAX_REC_SIZE. This record size is expressed in STORAGE_UNITs. MAX_RECORD_SIZE is defined in SYSTEM and can be changed by a program before instantiating DIRECT_IO or SEQUENTIAL_IO to provide an upper limit on the record size. For DIRECT_IO, the maximum size supported is 1024 * 1024 * STORAGE_UNIT bits. SEQUENTIAL_IO imposes no limit on MAX_REC_SIZE. DIRECT_IO can be instantiated with unconstrained types, but each element will be padded out to the maximum possible for that type or to SYSTEM.MAX_REC_SIZE, whichever is smaller.

F.11. 'Main' Programs

A 'main' program must be a non-generic subprogram that is either a procedure or a function returning STANDARD.INTEGER. While a 'main' program may not be a generic subprogram, it may, however, be an instantiation of a generic subprogram.

F.12. Machine Code Insertions

Machine code insertions are supported. See machine_code.a in the *standard* library.

F.13. Interrupt Entries

Interrupt entries are currently not supported.

F.14. Limits

The maximum line length (and thus the maximum identifier length) is 160 characters.

The maximum size of a statically sized record type is 4,000,000 * STORAGE_UNITs. A record type or array type declaration that exceeds these limits will generate a warning message. Also, a record component that depends on the (unconstrained) discriminant, e.g. string (1..discriminant); will generate a warning if the unconstrained maximum size is greater than 2_147_483_647.

In the absence of an explicit STORAGE_SIZE length specification, every task except the main program is allocated a fixed size stack of 40,960 STORAGE_UNITs. This is the value returned by T'STORAGE_SIZE for a task type T.

In the absence of an explicit STORAGE_SIZE length specification, the default collection size for an access type is 100,000 STORAGE_UNITs. This is the value returned by T'STORAGE_SIZE for an access type T.

Declared object size is limited only by available virtual space for the process.

APPENDIX C

TEST PARAMETERS

Certain tests in the ACVC make use of implementation-dependent values, such as the maximum length of an input line and invalid file names. A test that makes use of such values is identified by the extension .TST in its file name. Actual values to be substituted are represented by names that begin with a dollar sign. A value must be substituted for each of these names before the test is run. The values used for this validation are given below.

Name and Meaning	Value
$ACC_SIZE An integer literal whose value is the number of bits sufficient to hold any value of an access type.	32
$BIG_ID1 An identifier the size of the maximum input line length which is identical to $BIG_ID2 except for the last character.	(1..159 => 'A', 160 => '1')
$BIG_ID2 An identifier the size of the maximum input line length which is identical to $BIG_ID1 except for the last character.	(1..159 => 'A', 160 => '2')
$BIG_ID3 An identifier the size of the maximum input line length which is identical to $BIG_ID4 except for a character near the middle.	(1..80 => 'A', 81 => '3', 82..160 => 'A')

TEST PARAMETERS

Name and Meaning	Value
$BIG_ID4 An identifier the size of the maximum input line length which is identical to $BIG_ID3 except for a character near the middle.	(1..80 => 'A', 81 => '4', 82..160 => 'A')
$BIG_INT_LIT An integer literal of value 298 with enough leading zeroes so that it is the size of the maximum line length.	(1..157 => '0', 158..160 => "298")
$BIG_REAL_LIT A universal real literal of value 690.0 with enough leading zeroes to be the size of the maximum line length.	(1..154 => '0', 155..160 => "69.0E1")
$BIG_STRING1 A string literal which when catenated with BIG_STRING2 yields the image of BIG_ID1.	(1 => '"', 2..81 => 'A', 82 => '"')
$BIG_STRING2 A string literal which when catenated to the end of BIG_STRING1 yields the image of BIG_ID1.	(1 => '"', 2..80 => 'A', 81 => '1', 82 => '"');
$BLANKS A sequence of blanks twenty characters less than the size of the maximum line length.	(1..140 => ' ')
$COUNT_LAST A universal integer literal whose value is TEXT_IO.COUNT'LAST.	9_223_372_036_854_775_807
$DEFAULT_MEM_SIZE An integer literal whose value is SYSTEM.MEMORY_SIZE.	33_554_432
$DEFAULT_STOR_UNIT An integer literal whose value is SYSTEM.STORAGE_UNIT.	8

Name and Meaning	Value
$DEFAULT_SYS_NAME The value of the constant SYSTEM.SYSTEM_NAME.	ELXSI_4.3BSD 12.1
$DELTA_DOC A real literal whose value is SYSTEM.FINE_DELTA.	2.0 ** -31
$FIELD_LAST A universal integer literal whose value is TEXT_IO.FIELD'LAST.	9_223_372_036_854_775_807
$FIXED_NAME The name of a predefined fixed-point type other than DURATION.	NO_SUCH_FIXED_TYPE
$FLOAT_NAME The name of a predefined floating-point type other than FLOAT, SHORT_FLOAT, or LONG_FLOAT.	NO_SUCH_FLOAT_TYPE
$GREATER_THAN_DURATION A universal real literal that lies between DURATION'BASE'LAST and DURATION'LAST or any value in the range of DURATION.	131071.99993
$GREATER_THAN_DURATION_BASE_LAST A universal real literal that is greater than DURATION'BASE'LAST.	2147484.000
$HIGH_PRIORITY An integer literal whose value is the upper bound of the range for the subtype SYSTEM.PRIORITY.	7
$ILLEGAL_EXTERNAL_FILE_NAME1 An external file name which contains invalid characters.	/bad/file/BAD_CHARACTER*^
$ILLEGAL_EXTERNAL_FILE_NAME2 An external file name which is too long.	/bad/file/ NO_NAME_IS_TOO_LONG_BUT_IT_WONT_LIKE_THIS
$INTEGER_FIRST A universal integer literal whose value is INTEGER'FIRST.	-9223372036854775808

TEST PARAMETERS

Name and Meaning	Value
$INTEGER_LAST A universal integer literal whose value is INTEGER'LAST.	9223372036854775807
$INTEGER_LAST_PLUS_1 A universal integer literal whose value is INTEGER'LAST + 1.	9223372036854775808
$LESS_THAN_DURATION A universal real literal that lies between DURATION'BASE'FIRST and DURATION'FIRST or any value in the range of DURATION.	-131072.00000
$LESS_THAN_DURATION_BASE_FIRST A universal real literal that is less than DURATION'BASE'FIRST.	-2147484.000
$LOW_PRIORITY An integer literal whose value is the lower bound of the range for the subtype SYSTEM.PRIORITY.	0
$MANTISSA_DOC An integer literal whose value is SYSTEM.MAX_MANTISSA.	31
$MAX_DIGITS Maximum digits supported for floating-point types.	15
$MAX_IN_LEN Maximum input line length permitted by the implementation.	160
$MAX_INT A universal integer literal whose value is SYSTEM.MAX_INT.	9223372036854775807
$MAX_INT_PLUS_1 A universal integer literal whose value is SYSTEM.MAX_INT+1.	9_223_372_036_854_775_808
$MAX_LEN_INT_BASED_LITERAL A universal integer based literal whose value is 2#11# with enough leading zeroes in the mantissa to be MAX_IN_LEN long.	(1..2 => "2:", 3..157 => '0', 158..160 => "11:")

Name and Meaning	Value
$MAX_LEN_REAL_BASED_LITERAL A universal real based literal whose value is 16:F.E: with enough leading zeroes in the mantissa to be MAX_IN_LEN long.	(1..3 => "16:", 4..156 => '0', 157..160 => "F.E:")
$MAX_STRING_LITERAL A string literal of size MAX_IN_LEN, including the quote characters.	(1 => '"', 2..159 => 'A', 160 => '"')
$MIN_INT A universal integer literal whose value is SYSTEM.MIN_INT.	-9223372036854775808
$MIN_TASK_SIZE An integer literal whose value is the number of bits required to hold a task object which has no entries, no declarations, and "NULL;" as the only statement in its body.	32
$NAME A name of a predefined numeric type other than FLOAT, INTEGER, SHORT_FLOAT, SHORT_INTEGER, LONG_FLOAT, or LONG_INTEGER.	NO_SUCH_INTEGER_TYPE
$NAME_LIST A list of enumeration literals in the type SYSTEM.NAME, separated by commas.	ELXSI_4.3BSD 12.1, ELXSI_ATT
$NEG_BASED_INT A based integer literal whose highest order nonzero bit falls in the sign bit position of the representation for SYSTEM.MAX_INT.	16#FFFFFFFFFFFFFFFE#
$NEW_MEM_SIZE An integer literal whose value is a permitted argument for pragma MEMORY_SIZE, other than $DEFAULT_MEM_SIZE. If there is no other value, then use $DEFAULT_MEM_SIZE.	67_108_864

TEST PARAMETERS

Name and Meaning	Value
$NEW_STOR_UNIT An integer literal whose value is a permitted argument for pragma STORAGE_UNIT, other than $DEFAULT_STOR_UNIT. If there is no other permitted value, then use value of SYSTEM.STORAGE_UNIT.	8
$NEW_SYS_NAME A value of the type SYSTEM.NAME, other than $DEFAULT_SYS_NAME. If there is only one value of that type, then use that value.	ELXSI_ATT
$TASK_SIZE An integer literal whose value is the number of bits required to hold a task object which has a single entry with one 'IN OUT' parameter.	1024
$TICK A real literal whose value is SYSTEM.TICK.	0.01

APPENDIX D

WITHDRAWN TESTS

Some tests are withdrawn from the ACVC because they do not conform to the Ada Standard. The following 44 tests had been withdrawn at the time of validation testing for the reasons indicated. A reference of the form AI-ddddd is to an Ada Commentary.

a. E28005C: This test expects that the string "-- TOP OF PAGE. --63" of line 204 will appear at the top of the listing page due to a pragma PAGE in line 203; but line 203 contains text that follows the pragma, and it is this text that must appear at the top of the page.

b. A39005G: This test unreasonably expects a component clause to pack an array component into a minimum size (line 30).

c. B97102E: This test contains an unintended illegality: a select statement contains a null statement at the place of a selective wait alternative (line 31).

d. C97116A: This test contains race conditions, and it assumes that guards are evaluated indivisibly. A conforming implementation may use interleaved execution in such a way that the evaluation of the guards at lines 50 & 54 and the execution of task CHANGING_OF_THE_GUARD results in a call to REPORT.FAILED at one of lines 52 or 56.

e. BC3009B: This test wrongly expects that circular instantiations will be detected in several compilation units even though none of the units is illegal with respect to the units it depends on; by AI-00256, the illegality need not be detected until execution is attempted (line 95).

f. CD2A62D: This test wrongly requires that an array object's size be no greater than 10 although its subtype's size was specified to be 40 (line 137).

g. CD2A63A..D, CD2A66A..D, CD2A73A..D, and CD2A76A..D (16 tests): These

tests wrongly attempt to check the size of objects of a derived type (for which a 'SIZE length clause is given) by passing them to a derived subprogram (which implicitly converts them to the parent type (Ada standard 3.4:14)). Additionally, they use the 'SIZE length clause and attribute, whose interpretation is considered problematic by the WG9 ARG.

h. CD2A81G, CD2A83G, CD2A84M..N, and CD50110 (5 tests): These tests assume that dependent tasks will terminate while the main program executes a loop that simply tests for task termination; this is not the case, and the main program may loop indefinitely (lines 74, 85, 86, 96, and 58, respectively).

i. CD2B15C and CD7205C: These tests expect that a 'STORAGE_SIZE length clause provides precise control over the number of designated objects in a collection; the Ada standard 13.2:15 allows that such control must not be expected.

j. CD2D11B: This test gives a SMALL representation clause for a derived fixed-point type (at line 30) that defines a set of model numbers that are not necessarily represented in the parent type; by Commentary AI-00099, all model numbers of a derived fixed-point type must be representable values of the parent type.

k. CD5007B: This test wrongly expects an implicitly declared subprogram to be at the address that is specified for an unrelated subprogram (line 303).

l. ED7004B, ED7005C..D, and ED7006C..D (5 tests): These tests check various aspects of the use of the three SYSTEM pragmas; the AVO withdraws these tests as being inappropriate for validation.

m. CD7105A: This test requires that successive calls to CALENDAR.CLOCK change by at least SYSTEM.TICK; however, by Commentary AI-00201, it is only the expected frequency of change that must be at least SYSTEM.TICK--particular instances of change may be less (line 29).

n. CD7203B and CD7204B: These tests use the 'SIZE length clause and attribute, whose interpretation is considered problematic by the WG9 ARG.

o. CD7205D: This test checks an invalid test objective: it treats the specification of storage to be reserved for a task's activation as though it were like the specification of storage for a collection.

p. CE2107I: This test requires that objects of two similar scalar types be distinguished when read from a file--DATA_ERROR is expected to be raised by an attempt to read one object as of the other type. However, it is not clear exactly how the Ada standard 14.2.4:4 is to be interpreted; thus, this test objective is not considered valid (line 90).

q. CE3111C: This test requires certain behavior, when two files are associated with the same external file, that is not required by the Ada standard.

r. CE3301A: This test contains several calls to END_OF_LINE and END_OF_PAGE that have no parameter: these calls were intended to specify a file, not to refer to STANDARD_INPUT (lines 103, 107, 118, 132, and 136).

s. CE3411B: This test requires that a text file's column number be set to COUNT'LAST in order to check that LAYOUT_ERROR is raised by a subsequent PUT operation. But the former operation will generally raise an exception due to a lack of available disk space, and the test would thus encumber validation testing.